# Boxed Permutation Pattern Matching

**Mika Amit[1], Philip Bille[2], Patrick Hagge Cording[2], Inge Li Gørtz[2], and Hjalte Wedel Vildhøj[2]**

1 University of Haifa, Department of Computer Science
2 Technical University of Denmark, DTU Compute

## Abstract

Given permutations $T$ and $P$ of length $n$ and $m$, respectively, the Permutation Pattern Matching problem asks to find all $m$-length subsequences of $T$ that are order-isomorphic to $P$. This problem has a wide range of applications but is known to be NP-hard. In this paper, we study the special case, where the goal is to only find the *boxed subsequences* of $T$ that are order-isomorphic to $P$. This problem was introduced by Bruner and Lackner who showed that it can be solved in $O(n^3)$ time. Cho et al. [CPM 2015] gave an $O(n^2 m)$ time algorithm and improved it to $O(n^2 \log m)$. In this paper we present a solution that uses only $O(n^2)$ time. In general, there are instances where the output size is $\Omega(n^2)$ and hence our bound is optimal. To achieve our results, we introduce several new ideas including a novel reduction to 2D offline dominance counting. Our algorithm is surprisingly simple and straightforward to implement.

**1998 ACM Subject Classification** F.2.2 Computations and discrete structures, Geometrical problems and computations, Pattern matching. G.2.1 Combinatorial algorithms, Permutations and combinations

**Keywords and phrases** Permutation, Subsequence, Pattern Matching, Order Preserving, Boxed Mesh Pattern

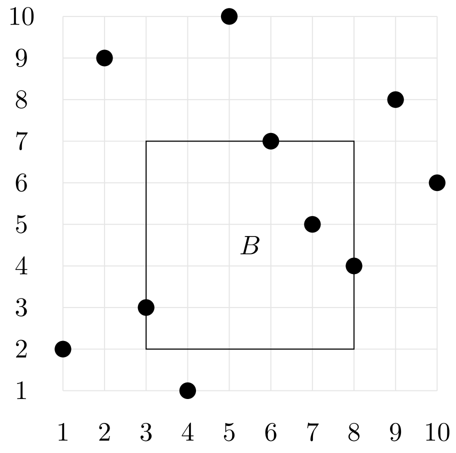
**Digital Object Identifier** 10.4230/LIPIcs.xxx.yyy.p

## 1 Introduction

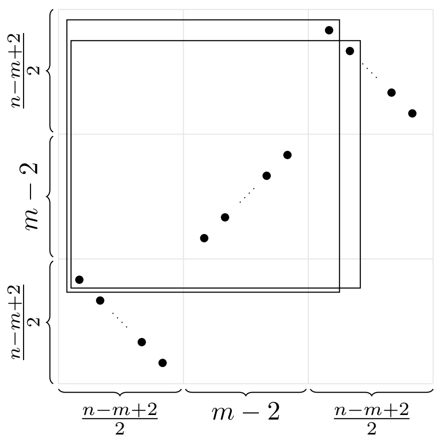
Consider a permutation $T = (t_1, t_2, \ldots, t_n)$ represented in the plane as the set of points $\{(1, t_1), (2, t_2), \ldots, (n, t_n)\}$. An axis-aligned *box* $B = (x_{\min}, x_{\max}, y_{\min}, y_{\max})$ that contains $|B| = k$ points induces a permutation $\sigma(B)$ of the integers $1, \ldots, k$. For example, the box shown in Figure 1a induces the permutation $\sigma(B) = (1, 4, 3, 2)$. Given $T$ and a permutation $P = (p_1, p_2, \ldots, p_m)$ (the pattern), the *boxed permutation pattern matching problem* is to output all boxes where $\sigma(B) = P$. If two boxes contain the same set of points, we consider them the same.

We view boxed permutation pattern matching as a natural 2D computational geometry problem, but it can also be seen and motivated as a generalization of *order-preserving pattern matching* (also known as *consecutive permutation pattern matching*). In this one-dimensional string matching problem the goal is to output all substrings of $T$ that are *order-isomorphic* to $P$. Order-preserving pattern matching has recently received a lot of attention (see e.g., [9, 10, 12, 13, 15, 17, 18, 20]) as it is a natural generalization of classic exact string matching, and since it can be used to search for trends in time series such as stock prices, music or weather data, etc.

Boxed permutation pattern matching solves order-preserving pattern matching if we only output the boxes of the form $B = (x_{\min}, x_{\min} + m - 1, -\infty, \infty)$ where $\sigma(B) = P$. However,



**(a)** A permutation $T = (2,9,3,1,10,7,5,4,8,6)$ and a box $B = (3,8,2,7)$ with $\sigma(B) = (1,4,3,2)$.



**(b)** A permutation $T$ with many occurrences of the pattern $P = (1,2,\ldots,m)$, $2 \le m \le n$. Precisely, $(n-m+2)^2/4$ boxes satisfy $\sigma(B) = P$, the figure shows two of them.

**Figure 1**

contrary to order-preserving pattern matching, which can be solved in $\tilde{O}(n)$ time by KMP-like algorithms [18,20], boxed permutation pattern matching requires $\Omega(n^2)$ time in general, as there are instances with $\Omega(n^2)$ occurrences of $P$ (see e.g. Figure 1b).

In this paper we present the first algorithm that solves boxed permutation pattern matching in optimal time, i.e., $O(n^2)$.

## 1.1 Previous and Related Work

Boxed permutation pattern matching was introduced by Bruner and Lackner [7] under the name *boxed-mesh permutation pattern matching*. They gave a simple $O(n^3)$ time algorithm. Recently, Cho et al. [11] presented a faster $O(n^2 m)$-time algorithm and showed how to improve it to $O(n^2 \log m)$ time. Bruner and Lackner, as well as Cho et al., defined the problem in terms of subsequences, but we note that our geometric definition of the problem is equivalent.

Boxed permutation pattern matching is one of a few special cases of *permutation pattern matching* that can be solved in polynomial time. This problem, which is known to be NP-hard [4], asks to output all subsequences of $T$ that are order-isomorphic to $P$. Due to the many applications of permutation pattern matching a vast amount of research has studied its generalizations (e.g., vincular [3,19], bivincular [5] and mesh [6] patterns) and special cases (e.g. boxed mesh [2,11] and consecutive patterns [9,12,13,17,18,20] or patterns with certain combinatorial properties [1,16]). We refer the reader to Bruner and Lackner [7] for definitions and a comprehensive in-depth overview of previous work.

## 1.2 Our Result

We show the following result.

▶ **Theorem 1.** *Boxed permutation pattern matching can be solved in $O(n^2)$ time.*

As there are instances with $\Omega(n^2)$ outputs (see Figure 1b), this time bound is optimal.

Our algorithm improves the $O(n^2 \log m)$-time algorithm by Cho et al. [11]. The $\log m$ factor in their time bound comes from their use of an order statistics tree with update time $O(\log m)$ to represent a box. We observe that plugging in the more efficient data structure by Pătraşcu and Thorup [21] immediately improves their time complexity to $O(n^2 \log m / \log \log n)$. However, as their solution inherently requires dynamic rank (or select) queries on the $\Omega(m)$ points inside a box, we cannot hope to further improve the time bound with this approach due to lower bounds on dynamic rank and select queries [14, 21].

We circumvent this apparent problem as follows: Instead of representing a box by the $\Omega(m)$ points it contains, we represent it in constant space by storing its four sides. Hence we can easily update the representation in constant time whenever we add a new point to the box. The challenge is to efficiently check *if* a point can be added or not. We show that implementing this check can be reduced to *2D offline dominance counting* on $n$ subproblems of size $O(n)$. Solving these subproblems individually using the state-of-the-art $O(n\sqrt{\log n})$-time algorithm for 2D offline dominance problem by Chan and Pătraşcu [8] leads to an $O(n^2\sqrt{\log n})$-time solution for boxed permutation pattern matching. To get $O(n^2)$ time, we exploit the close relationship of the $n$ subproblems, and show that it suffices to solve just a single of these subproblems.

Our final algorithm is surprisingly simple and straightforward to implement, as it only relies on a few lookup tables and uses no complicated supporting data structures.

## 2 Preliminaries

We start by giving some necessary definitions and combinatorial properties. Let $P_k$, $1 \leq k \leq m$, be the permutation of the integers $1, \ldots, k$ induced by the prefix $(p_1, p_2, \ldots, p_k)$ of $P$ (see Figure 1a). For a box $B = (i, j, y_{\min}, y_{\max})$ we define its size $|B|$ to be the number of points it contains. We use $\cdot$ to denote if one or more sides of $B$ are unspecified, i.e., $(i, j, \cdot, \cdot)$ denotes an arbitrary box with $x_{\min} = i$ and $x_{\max} = j$. We say that $B = (i, j, \cdot, \cdot)$ is *anchored* if it includes the point $(i, t_i)$ as its left-most point. Furthermore, we say that $B$ is a *prefix box* if $B$ is anchored and $\sigma(B) = P_{|B|}$, in which case we also say that $B$ *matches* the prefix of $P$ of size $|B|$. We need the following lemma, which is similar to Lemma 2 in [11].

▶ **Lemma 2.** *For fixed integers $1 \leq i \leq j \leq n$ and $1 \leq k \leq m$, there is at most one prefix box $B = (i, j, y_{\min}, y_{\max})$ of size $k$.*

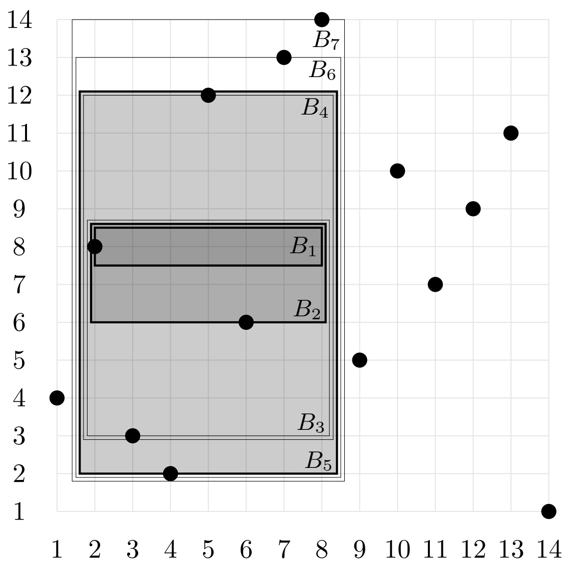
**Proof.** Let $i, j$ and $k$ be fixed, and let $s$ and $l$ denote the number of elements of $\{p_2, \ldots, p_k\}$ that are smaller and larger than $p_1$, respectively. A prefix box $(i, j, \cdot, \cdot)$ of size $k$ must contain the first $s$ points below $t_i$ and the first $l$ points above $t_i$, and hence it is unique. ◀

The proof of the lemma uses the fact that for fixed $i, j, k$ there is a unique candidate box $(i, j, \cdot, \cdot)$ that can match $P_k$. Figure 2b shows these candidate boxes for $k = 1, \ldots, m$ (and $i$ and $j$ fixed). Observe that only the prefixes $P_1$,$P_2$ and $P_5$ are matched, and that the boxes are nested, i.e., $B_{i-1}$ is contained in $B_i$.

Given a prefix box $B = (i, j, \cdot, \cdot)$ its *preceding prefix box* is the largest prefix box $B' = (i, j, \cdot, \cdot)$ smaller than $B$. Observe that the preceding prefix box can be obtained by removing a certain number of points from above and a certain number of points from below, and note that these two numbers only depend on the size of $B$. Consequently, for a prefix box $B$ of size $k = 2, \ldots, m$, we let $(a_k, b_k)$ be the number of points that needs to be removed from $B$ from above and below, respectively, to obtain the preceding prefix box of $B$. We will show how to compute these numbers later.

$$
\begin{aligned}
P = P_7 &= 4,2,1,6,3,5,7\\
P_6 &= 4,2,1,6,3,5\\
P_5 &= 4,2,1,5,3\\
P_4 &= 3,2,1,4\\
P_3 &= 3,2,1\\
P_2 &= 2,1\\
P_1 &= 1
\end{aligned}
$$

**(a)** The permutations induced by the prefixes of the pattern $P = (4, 2, 1, 6, 3, 5, 7)$.



**(b)** The unique boxes $(2, 8, \cdot, \cdot)$ that can match $P_1, \ldots, P_7$. In this specific case only the three bold boxes $B_1, B_2, B_5$ are prefix boxes and match the corresponding prefix of $P$.

**Figure 2**

▶ **Example 3.** In Figure 2b there are three prefix boxes on $(i, j) = (2, 8)$: $B_5$, $B_2$ and $B_1$. The preceding prefix box of $B_5$ is $B_2$ and to obtain it we need to remove one point from above and two from below, so $(a_5, b_5) = (1, 2)$. Similarly, $B_1$ is the preceding prefix box of $B_2$ and to obtain it, we need to remove one point from below, so $(a_2, b_2) = (0, 1)$.

## 3 The Flattening Box Algorithm

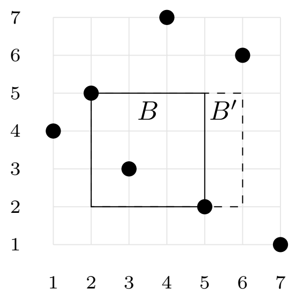
At a high level the algorithm of Cho et al. [11] and our new algorithm can both be seen as implementations of an abstract algorithm, which we call *the flattening box algorithm*. The name comes from the fact that it examines boxes of decreasing height and increasing length. In this section we give a geometric exposition of this abstract algorithm, and in the next section we elaborate on how to implement the three primitives it needs.

Let $B_{\max}(i, j)$ be the largest prefix box $(i, j, \cdot, \cdot)$, e.g., in Figure 2b, we have that $B_{\max}(2, 8) = B_5$. We also refer to $B_{\max}(i, j)$ as the *maximum prefix box* on $(i, j)$. Note that by Lemma 2, $B_{\max}(i, j)$ is unique, and observe that if $|B_{\max}(i, j)| = m$ then it corresponds to an occurrence of $P$.

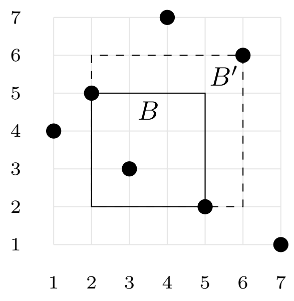
Recall that to solve the boxed permutation pattern matching problem, we need to find all boxes $B$ s.t. $\sigma(B) = P$. The flattening box algorithm actually solves the slightly more general problem of computing $B_{\max}(i, j)$ for all $1 \le i \le j \le n$. We do this in $n$ iterations $(i = 1, \ldots, n)$, and in each iteration we compute $B_{\max}(i, j)$ for all $j = i, \ldots, n$. The main idea is to compute $B_{\max}(i, j)$ as a so-called *extension* of another prefix box $(i, j - 1, \cdot, \cdot)$.

### 3.1 Extensions of Prefix Boxes

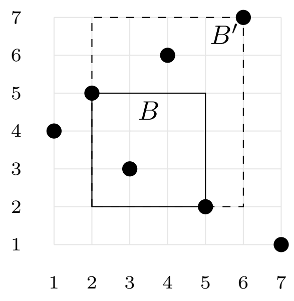
We say that a prefix box $B = (i, j, y_{\min}, y_{\max})$ has an *empty extension* $B' = (i, j + 1, y_{\min}, y_{\max})$ if $B'$ is a prefix box also of size $|B|$. Note that this means $B'$ contains exactly



**(a)** An empty extension



**(b)** An increasing extension



**(c)** Not an extension

**Figure 3** Illustrating different extensions of a prefix box $B$ of the pattern $P$ shown in Figure 1a. (a) $B'$ is an empty extension of $B$, since $\sigma(B) = P_{|B|} = \sigma(B')$. (b) $B'$ is an increasing extension of $B$, since $\sigma(B) = P_{|B|}$ and $\sigma(B') = P_{|B|+1}$. (c) $B'$ is not an extension of $B$.

the same points as $B$. See Figure 3a for an example. Moreover, we say that $B$ has an *increasing extension* $B' = (i, j+1, \min(y_{\min}, t_{j+1}), \max(y_{\max}, t_{j+1}))$, if $B'$ is a prefix box of size $|B|+1$, i.e., $\sigma(B') = P_{|B|+1}$. Note, here $B'$ is the box obtained by extending $B$ to include the point $(j+1, t_{j+1})$. See Figure 3b for an example.
The following lemma shows that we can compute the prefix boxes of the form $(i, j, \cdot, \cdot)$ as the extensions of the prefix boxes of the form $(i, j-1, \cdot, \cdot)$.

▶ **Lemma 4.** *Let $1 \leq i < j \leq n$. Any prefix box $B = (i, j, \cdot, \cdot)$ is an extension of a prefix box $B' = (i, j-1, \cdot, \cdot)$.*

**Proof.** Let $B = (i, j, y_{\min}, y_{\max})$, $j > i$, be a prefix box and consider the box $B' = (i, j-1, y_{\min}, y_{\max})$. Clearly, $B'$ is also a prefix box, and if $|B'| = |B| - 1$, $B$ is an increasing extension of $B'$, and otherwise $|B'| = |B|$ and $B$ is an empty extension of $B'$. ◀

Let $B_{\text{ext}}(i, j) = (i, j, \cdot, \cdot)$ denote the largest prefix box that has an extension. We then have the following important corollary, which we will use for computing $B_{\max}(i, j)$.

▶ **Corollary 5.** *$B_{\max}(i, j)$ is an (increasing or empty) extension of $B_{\text{ext}}(i, j-1)$.*

▶ **Example 6.** In Figure 2b there are three prefix boxes $B_5, B_2$ and $B_1$ on $(i, j) = (2, 8)$. $B_5$ has no extension, $B_2$ has both an increasing and an empty extension, and $B_1$ has an empty extension. Consequently, $B_{\text{ext}}(i, j) = B_2$, and thus the largest prefix box $B_{\max}(i, j+1)$ is the increasing extension of $B_2$, i.e., $(2, 9, 5, 8)$, matching the prefix $P_3$.

## 3.2 The Abstract Algorithm

Our goal is to compute $B_{\max}(i, j)$ assuming we have already computed $B_{\max}(i, j-1)$. According to Corollary 5, we need to first find $B_{\text{ext}}(i, j-1)$. Note that as shown by Example 6, $B_{\text{ext}}(i, j-1)$ is not necessarily equal to $B_{\max}(i, j-1)$. However, we can find $B_{\text{ext}}(i, j-1)$ as follows: Starting with $B_{\max}(i, j-1)$ (which we have computed), we check each of the prefix boxes $(i, j-1, \cdot, \cdot)$ in decreasing order of their size. The first one, which has an extension (increasing or empty) gives us $B_{\text{ext}}(i, j-1)$, and hence also $B_{\max}(i, j)$. Algorithm 1 shows this approach, assuming that we have available a function `precedingPrefixBox`, which takes a prefix box $B = (i, j-1, \cdot, \cdot)$ and returns the largest prefix box $(i, j-1, \cdot, \cdot)$ smaller than $B$.

```
Input  : Permutations $T = (t_1, \ldots, t_n)$ and $P = (p_1, \ldots, p_m)$
Output: All boxes $B$ s.t. $\sigma(B) = P$
for $i \leftarrow 1$ to $n$ do
    $B_{\max}(i, i) \leftarrow (i, i, t_i, t_i)$
    for $j \leftarrow i + 1$ to $n$ do
        $B \leftarrow B_{\max}(i, j - 1)$
        while $B_{\max}(i, j) =$ null do
            if $B$ has an increasing extension $B'$ then
                $B_{\max}(i, j) \leftarrow B'$
            else if $B$ has an empty extension $B'$ then
                $B_{\max}(i, j) \leftarrow B'$
            else
                $B \leftarrow$ precedingPrefixBox($B$)
            end
        end
        if $|B_{\max}(i, j)| = m$ then
            Output $B_{\max}(i, j)$                /* Found an occurrence of $P$ */
        end
    end
end
```

**Algorithm 1:** The Flattening Box Algorithm

## 4 Implementing the Algorithm

To implement the abstract algorithm we need to describe how to check if a prefix box $B$ has an increasing/empty extension, and how to obtain the preceding prefix box of $B$. We describe how to do this in the following sections.

### 4.1 Checking If a Prefix Box Can Be Extended

We can easily check in constant time whether a given prefix box $B = (i, j, y_{\min}, y_{\max})$ has an empty extension, since this is the case if and only if $t_{j+1} \notin [y_{\min}, y_{\max}]$.

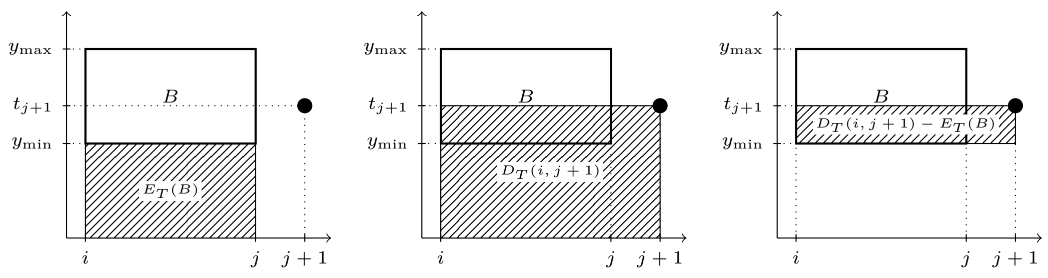
Hence in the remaining part of this section we focus on how to efficiently check whether $B$ has an increasing extension, which is a significantly more involved.

We need the following definitions. For a permutation $Q = (q_1, \ldots, q_{|Q|})$, we define $D_Q(i, j) = |(i, j, 1, q_j)|$, i.e., $D_Q(i, j)$ is the number of points $(l, q_l)$ where $i \le l \le j$ and $q_l \le q_j$ (see Figure 4b). Moreover, for a box $B = (i, j, y_{\min}, y_{\max})$, we define $E_Q(B) = |(i, j, 1, y_{\min} - 1)|$, i.e., $E_Q(B)$ is the number of points below $B$ (see Figure 4a).

The following lemma gives the property that we will use for checking if a prefix box $B$ has an increasing extension in constant time.

▶ **Lemma 7.** *A prefix box $B = (i, j, y_{\min}, y_{\max})$ has an increasing extension if and only if $D_T(i, j + 1) - E_T(B) = D_P(1, |B| + 1)$.*

**Proof.** We need the following simple *append* operation on permutations. Let $Q = (q_1, \ldots, q_k)$ be a permutation of the integers $1, \ldots, k$. The permutation obtained by *appending* an integer $1 \le r \le k + 1$ to $Q$ is the permutation $Q \cdot r = (q'_1, \ldots, q'_k, r)$ of the integers $1, \ldots, k + 1$, where for $1 \le i \le k$, $q'_i = q_i + 1$ if $q_i \ge r$ and $q'_i = q_i$ otherwise.



**(a)** $E_T(B)$ **(b)** $D_T(i,j+1)$ **(c)** $D_T(i,j+1) - E_T(B)$

**Figure 4** Illustrating the numbers used to decide if a prefix box $B$ has an increasing extension.

Now to prove the lemma, we start by observing that $P_{k+1} = P_k \cdot D_P(1,k+1)$ for $1 \le k \le m-1$. We need to show that the box $B' = (i, j+1, \min(t_{j+1}, y_{\min}), \max(t_{j+1}, y_{\max}))$ obtained by extending $B$ to include the point $(j+1, t_{j+1})$, induces the permutation $P_{|B|+1}$ if and only if $D_T(i,j+1) - E_T(B) = D_P(1,|B|+1)$. We consider the two cases $|B'| = |B|+1$ and $|B'| > |B|+1$ separately.

In the first case $|B'| = |B|+1$, which means $B'$, in addition to the points in $B$, includes only the point $(j+1, t_{j+1})$. It is not hard to see that $D_T(i,j+1) - E_T(B)$ counts the number of points $(l, t_l) \in B$ s.t. $t_l \le t_{j+1}$ (see Figure 4). Hence the induced permutation of $B'$ is

$$\sigma(B') = \sigma(B) \cdot (D_T(i,j+1) - E_T(B)) \ .$$

At the same time we have that $B$ has an increasing extension if and only if

$$\sigma(B') = P_{|B|+1} = P_{|B|} \cdot D_P(1,k+1) = \sigma(B) \cdot D_P(1,k+1) \ .$$

Combining the two equations yields the lemma.

In the other case $|B'| > |B|+1$. This means that $B'$ in addition to $(j+1, t_{j+1})$ also contains some other points (either above or below $B$). See Figure 3c for an example. Clearly $\sigma(B') \ne P_{|B|+1}$ and thus $B$ has no increasing extension in this case. To show that $D_T(i,j+1) - E_T(B) \ne D_P(1,|B|+1)$, observe that if points above $B$ were included $D_T(i,j+1) - E_T(B) > |B|+1$, and if points below $B$ were included $D_T(i,j+1) - E_T(B) \le 0$. Since $1 \le D_P(1,|B|+1) \le |B|+1$, we have that $D_T(i,j+1) - E_T(B) \ne D_P(1,|B|+1)$. ◀

In the following we describe how to efficiently obtain the values of $D_T(i,j+1)$, $E_T(B)$ and $D_P(1,|B|+1)$.

### 4.1.1 The value $D_P(1,|B|+1)$

Prior to running the algorithm, we preprocess a table of size $O(m)$ that stores the value $D_P(1,k)$ for $k = 2,\ldots,m$. Assuming we maintain the size of the current prefix box $B$ in the algorithm (which is straightforward), we can obtain $D_P(1,|B|+1)$ by a constant-time lookup. We describe how to compute the lookup table in $O(m\sqrt{\log m})$ time in Section 4.3.

### 4.1.2 The value $E_T(B)$

Recall that in the algorithm we consider the prefix boxes $(i,j,\cdot,\cdot)$ in decreasing order of their size until we find $B_{\text{ext}}(i,j)$. For each such prefix box $B$, we need the value of $E_T(B)$, i.e., the number of points below that prefix box. We maintain this number as we iterate $j$ from $i$ to $n$ as follows.

Initially (for $j = i$) there are no points below the box, so $E_T(B) = 0$. The number only changes in the following two cases: If $B$ does not have an extension, we consider the preceding prefix box of $B$, and hence $E_T(B)$ increases by $b_k$ (the number of points that are removed from below). The other case in which $E_T(B)$ changes is when $B$ is extended to $j+1$ as an empty extension and $t_{j+1} < y_{\min}$. In this case $E_T(B)$ increases by one.

### 4.1.3 The value $D_T(i, j+1)$

In the following assume that $i$ is fixed, corresponding to a single iteration of the outer-most loop of the algorithm. As we iterate $j$ from $i$ to $n$, we need the value $D_T(i, j)$. We compute these values for $j = i, \ldots, n$ in the beginning of iteration $i$ and store them in a table of size $O(n)$. Recall that $D_T(i, j)$ is the number of points with $x$ value between $i$ and $j$ and $y$-value below $t_j$. Hence we can compute the value $D_T(i, j)$ from $D_T(i-1, j)$ as follows.

$$D_T(i,j) = \begin{cases} D_T(i-1,j) - 1 & \text{if } t_{i-1} < t_j \\ D_T(i-1,j) & \text{otherwise} \end{cases} \tag{1}$$

Note that computing the table only takes $O(n)$ time, assuming we have the table $D_T(i-1, j)$. Moreover, there is no need to store the old table, so we only need $O(n)$ space over all iterations of the algorithm. However, in the very first iteration ($i = 1$), we need the table $D_T(1, j)$, $j = 1, \ldots, n$. We compute this table in $O(n\sqrt{\log n})$ time in the preprocessing phase of the algorithm as explained in Section 4.3.

## 4.2 Computing the Preceding Prefix Box

Recall that given a prefix box $B = (i, j, y_{\min}, y_{\max})$ of size $k$, its preceding prefix box $B'$ can be obtained by removing $a_k$ points from above, and $b_k$ points from below. We compute $(a_k, b_k)$ for all $k$ in the preprocessing phase as explained in Section 4.3.

To remove $a_k$ points from above, we decrement $y_{\max}$ in steps of one and keep track of how many points we have excluded. Note that when decrementing $y_{\max}$ we exclude a point from $B$ if and only if $i \le T^{-1}(y_{\max}) \le j$, where $T^{-1}$ is the *inverse permuation* of $T$, i.e., $T^{-1}(t_i) = i$. We remove the $b_k$ points from below by similarly incrementing $y_{\min}$ until $b_k$ points have been excluded.

We compute $T^{-1}$ in the preprocessing phase of the algorithm in $O(n)$ time.

## 4.3 Preprocessing the Lookup Tables

We now describe and analyze the necessary preprocessing of the text $T$ and the pattern $P$.

### 4.3.1 Preprocessing of the Text

For the text, we need two $O(n)$-size tables, its inverse permutation $T^{-1}$, and the table for $D_T(1, j)$, $j = 1, \ldots, n$. The inverse permutation is easily computed in $O(n)$ time.

Recall that $D_T(1, j)$ is the number of points with strictly smaller coordinates than $(j, t_j)$. The problem of computing this number for all $n$ points is known as *2D offline dominance counting*, and can be solved in $O(n\sqrt{\log n})$ time using the algorithm by Chan and Pătraşcu [8]. In fact, since we only need to compute this table for $i = 1$, we can afford to use the trivial $O(n^2)$-time algorithm as well.

### 4.3.2 Preprocessing of the Pattern

For the pattern we need two $O(m)$-size tables, the table for $D_P(1,k)$, $k = 1, \dots, m$, and the table storing the $(a_k, b_k)$ values for $k = 1, \dots, m$. Computing the table for $D_P(1,k)$ can be done in $O(m\sqrt{\log m})$ time exactly as we did for the text.

We will compute the $(a_k, b_k)$ values incrementally using an algorithm very similar to the flattening box algorithm. Recall that $(a_k, b_k)$ denote the number of points that must be removed from a prefix box $B$ of size $k$, from above and below, respectively, to obtain its preceding prefix box. Initially, we set $(a_1, b_1) = (0, 0)$. Let $T = P$, i.e., we treat the set of points $\{(1, p_1), \dots, (m, p_m)\}$ as the text. Now consider a box $B_k = (1, k, 1, m)$. This box is clearly a prefix box, since it matches $P_k$, and hence it is also the maximum prefix box of the form $(1, k, \cdot, \cdot)$. Let $B'_k$ denote the *second largest* prefix box on $(1, k, \cdot, \cdot)$, i.e., the preceding prefix box of $B_k$. The idea is to compute $B'_k$, for $k = 1, \dots, m$. It is easy to see that this will give us the $(a_k, b_k)$ values as the number of points above and below $B'_k$, respectively.

It follows from Lemma 2 that we can compute $B'_k$ from $B'_{k-1}$ as follows: Starting with $B'_{k-1}$ (which we have computed), we check each of the prefix boxes $(1, k-1, \cdot, \cdot)$ in decreasing order of their size. The first one, which has an extension (increasing or empty) of size less than $k$, gives us $B'_k$. Note that as we have already computed $(a_{k'}, b_{k'})$ for $k' < k$, we can compute the preceding prefix box of $B'_{k-1}$ (and any of its predecessors) by simply removing $(a_{k-1}, b_{k-1})$ points from above and below using exactly the same approach as in the flattening box algorithm (See Section 4.2). This requires that we also compute the inverse permutation of $P$ in $O(m)$ time and space.

The time for computing all $(a_k, b_k)$ values can be bounded by $O(m^2)$, for the same reason the flattening box algorithm runs in $O(n^2)$ time (See the next section).

## 5 Analysis

We now summarize the time analysis of our algorithm.

As already argued, we need $O(n\sqrt{\log n})$ time and $O(n)$ space for preprocessing the text, and preprocessing of the pattern takes $O(m^2)$ time and $O(m)$ space. To prove that the algorithm runs in $O(n^2)$ time, we show that a single iteration of the outer-most for-loop only takes $O(n)$ time. Checking if a prefix box can be extended only requires constant-time table lookups. To see that the total time spent computing preceding prefix boxes is $O(n)$, it suffices to note that once a point is excluded, it can never be included again. Hence the total time spent decrementing $y_{\max}$ and incrementing $y_{\min}$ is $O(n)$.

Consequently, the total time complexity is $O(n^2 + m^2) = O(n^2)$.

## 6 Open Problems

We have shown that boxed permutation pattern matching can be solved in $O(n^2)$ time, which is optimal. Our algorithm uses $O(n)$ space, which leaves open the problem of reducing the space to $O(m)$. The main challenge in doing this is the apparent absence of a suitable decomposition of a problem instance into $O(n^2/m^2)$ independent subproblems of size $O(m^2)$. We do believe that an $O(n^2)$ time and $O(m)$ space algorithm exists, but note that with our current techniques, it seems difficult to reduce the space to $O(m)$ without increasing the time to at least $O(n^2 \log\log n)$.

Another interesting direction for future work is the possibility of designing output-sensitive algorithms. That is, can an instance of boxed permutation pattern matching with *occ* occurrences of the pattern be solved in $O(n^{2-\epsilon} + occ)$ time, for some $\epsilon > 0$?

Finally, we note that indexing and approximate variants of boxed permutation pattern matching also have not been studied yet.

## References

1 M. H. Albert, R. E. Aldred, M. D. Atkinson, and D. A. Holton. Algorithms for pattern involvement in permutations. In *Algorithms and Computation*, pages 355–367. Springer, 2001.

2 S. Avgustinovich, S. Kitaev, and A. Valyuzhenich. Avoidance of boxed mesh patterns on permutations. *Discrete App. Math.*, 161(1):43–51, 2013.

3 Eric Babson and Einar Steingrımsson. Generalized permutation patterns and a classification of the mahonian statistics. *Sém. Lothar. Combin*, 44(B44b):547–548, 2000.

4 P. Bose, J. F. Buss, and A. Lubiw. Pattern matching for permutations. *Inform. Process. Lett.*, 65(5):277–283, 1998.

5 M. Bousquet-Mélou, A. Claesson, M. Dukes, and S. Kitaev. (2+ 2)-free posets, ascent sequences and pattern avoiding permutations. *J. Comb. Theory A*, 117(7):884–909, 2010.

6 P. Brändén and A. Claesson. Mesh patterns and the expansion of permutation statistics as sums of permutation patterns. *Electron. J. Combin*, 18(2):P5, 2011.

7 M. L. Bruner and M. Lackner. The computational landscape of permutation patterns. *CoRR*, abs/1301.0340, 2013.

8 T. M. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proc. 21st ACM-SIAM symposium on Discrete Algorithms*, pages 161–173. Society for Industrial and Applied Mathematics, 2010.

9 T. Chhabra and J. Tarhio. Order-preserving matching with filtration. In *Experimental Algorithms*, pages 307–314. Springer, 2014.

10 S. Cho, J. C. Na, K. Park, and J. S. Sim. A fast algorithm for order-preserving pattern matching. *Inform. Process. Lett.*, 115(2):397–402, 2015.

11 S. Cho, J. C. Na, and J. S. Sim. Improved algorithms for the boxed-mesh permutation pattern matching problem. In *Proc. 26th CPM*, pages 138–148. Springer, 2015.

12 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Waleń. Order-preserving incomplete suffix trees and order-preserving indexes. In *Proc. 20th SPIRE*, pages 84–95. Springer, 2013.

13 S. Faro and M. O. Külekci. Efficient algorithms for the order preserving pattern matching problem. *CoRR*, abs/1501.04001, 2015.

14 Michael Fredman and Michael Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proc. 21st STOC*, pages 345–354. ACM, 1989.

15 P. Gawrychowski and P. Uznański. Order-preserving pattern matching with k mismatches. In *Proc. 25th CPM*, pages 130–139. Springer, 2014.

16 L. Ibarra. Finding pattern matchings for permutations. *Inform. Process. Lett.*, 61(6):293–295, 1997.

17 J. Kim, A. Amir, J. C. Na, K. Park, and J. S. Sim. On representations of ternary order relations in numeric strings. In *Proc. 2nd ICABD*, pages 46–52, 2014.

18 J. Kim, P. Eades, R. Fleischer, S. H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order-preserving matching. *Theoret. Comput. Sci.*, 525:68–79, 2014.

19 S. Kitaev. *Patterns in permutations and words.* Springer Science & Business Media, 2011.

20 M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń. A linear time algorithm for consecutive permutation pattern matching. *Inform. Process. Lett.*, 113(12):430–433, 2013.

21 M. Patrascu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th FOCS*, pages 166–175. IEEE, 2014.